Flexible Object Encapsulation for Ambient-Oriented Programming[*]

Wolfgang De Meuter[1,2] Éric Tanter[3] Stijn Mostinckx[2]
Tom Van Cutsem[2] Jessie Dedecker[2]

[1] Laboratoire d'Informatique Fondamontale de Lille
Université des Sciences et Technologies de Lille, France

[2] Programming Technology Lab
Vrije Universiteit Brussel, Belgium

[3] Center for Web Research, DCC
University of Chile, Chile

{wdmeuter,smostinc,tvcutsem,jededeck}@vub.ac.be, etanter@dcc.uchile.cl

ABSTRACT

In the emerging field of Ambient Intelligence (AmI), software is deployed in wireless open networks of mobile devices. Such open networks require stringent security measures as unknown and untrusted hosts may join the network. In an object-oriented language, where objects are distributed and moved across the network, it thus becomes important to be able to *enforce* object encapsulation. In contemporary object-oriented programming languages, powerful operations such as object extension (inheritance), cloning and reflection, are typically provided via omnipotent language operators that fail to uphold object encapsulation, because they can be applied without the explicit consent of the concerned object. This paper formulates a language design principle *–extreme encapsulation–* that precludes the use of such harmful operators, and proposes a corresponding language feature *–method attributes–* that makes it possible to provide the flexibility of object extension, cloning and reflection without compromising on object encapsulation. Although some existing object-based languages can be said to support extreme encapsulation, our contribution is to support it in a delegation-based, prototype-based language named ChitChat.

1. INTRODUCTION

Software development for mobile devices has become a major issue with the advent of *mobile networks*. Mobile networks surround a mobile device equipped with wireless technology and are demarcated dynamically as users move around. Mobile networks turn the applications running on mobile devices from mere isolated programs into smart applications that can cooperate with their environment. As such, mobile networks take us one step closer to the world of ubiquitous computing envisioned by Weiser [29]: a world where (wireless) technology is gracefully integrated into the everyday lives of its users. Recently, this vision has been termed *Ambient Intelligence* (AmI for short) by the European Council's IST Advisory Group [13].

Although low-level technologies for programming software for mobile networks have matured, current programming languages lack abstractions to deal with the specificities of mobile networks. This has led some of the authors to propose a new *Ambient-Oriented Programming* paradigm [9, 5] (AmOP for short), which consists of programming languages that incorporate the specificities of mobile networks at the very heart of their basic computational steps. We identify three axes of programming language design for AmI: concurrency, distribution, and the object model. Some of the authors have already proposed language abstractions for the two first axes [5, 8]. This paper focuses on the third axis, the object model, which specifies what objects are and which operations are supported by the language.

In mobile computing applications for Ambient Intelligence, devices form ad hoc networks with other devices that are dynamically discovered in the environment. Since every encountered device cannot necessarily be trusted, proper security measures are required when objects are interchanged. As stated by Vitek *et al.*, an object that is copied or moved should be protected from abuse by malicious objects in the new host, and a host should be able to hand out access to its resources on a very selective basis, in order to protect it from malicious objects [28]. Our approach to improve the security of object-oriented applications is based on the principled use of object encapsulation [22]. We propose a language design principle, named *extreme encapsulation*, to ensure that the integrity of objects can be upheld at any time.

Most contemporary object-oriented languages are, however, incompatible with the extreme encapsulation principle, because they typ-

[*]É. Tanter is financed by the Milenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile, and the ITCC Chile-Korea. S. Mostinckx is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). T. Van Cutsem and J. Dedecker are research assistants of the Fund for Scientific Research Flanders, Belgium (F.W.O.).

ically provide possibly harmful *operations* such as object extension, cloning or reflection, via *language operators* that can be applied to objects without their explicit consent. Extreme encapsulation can be trivially achieved if these operations are simply abolished from the language. However, this would severely restrict the language because the harmful operations have real practical value. This paper therefore proposes a language feature that makes it possible to provide these operations without resorting to omnipotent, encapsulation-breaching operators. Indeed, we reconcile object extension, cloning and reflection with extreme encapsulation through the introduction of *method attributes*: methods can be annotated with attributes leading to different evaluation semantics, allowing them to replace the harmful language operators.

We proceed as follows: in Section 2 we present general considerations of object-oriented languages for open networks, which motivate the use of an object-based language, and identify the different operations that a full-fledged object model should support. Section 3 analyzes the conflicts between these operations and the preservation of object encapsulation. Then, in Section 4, we formulate the principle of *extreme encapsulation* that AmOP languages should respect. We present our model of extremely-encapsulated prototypes in Section 5, explaining and illustrating how method attributes allow for the provision of object extension, cloning and reflection without compromising on extreme encapsulation. Section 6 discusses related work, and Section 7 concludes with perspectives for future research.

2. OBJECT-ORIENTED LANGUAGES FOR OPEN NETWORKS

In this section we first argue in favor of object-based programming languages in the context of Ambient-Oriented Programming. We then identify the main features of object-oriented languages that characterize a full-fledged object model.

2.1 Object-based Languages for Open Networks

The field of object-oriented languages is separated into two kinds of object models: those based on classes and those based solely on objects. Representatives of the former are Smalltalk [11] and Java [12]. Languages employing a pure object-based design are Self [27] and earlier versions of Javascript. In this paper, we argue in favor of object-based models to program distributed objects. Conceptually, this model is simpler than its class-based counterpart since one only has to deal with a single kind of entities, namely objects. As a consequence, all language features investigated in this paper –extension, cloning and reflection– are directly defined upon objects.

Another argument in favor of object-based languages is that objects do not implicitly depend on a class structure. In distributed programming, this dependency requires a class-based language to replicate classes over different hosts in the network when objects are passed as parameter or return value during a remote method invocation. Such extensive class copying among hosts is problematic because, from a conceptual point of view, there ought to be only one single version of a class on the network, containing the class variables and method implementations shared by all instances. This sharing relation between the instances is established at object creation time and remains *implicit* throughout their lifetime. However, when each host has its "own" copy of the class, the copies may evolve independently, such that two instances of the same conceptual class may unexpectedly exhibit different behavior because they have a different concrete class [25]. Because of these independent changes to class copies, the implicit object-class relation becomes explicitly detectable.

Object-based distributed programming languages will, of course, suffer from the consequences of code duplication as well. The difference lies in the way programmers can deal with the problem. Most existing class-based languages do not offer programmers the means to deal with multiple copies of the same class, because of the implicitness of the instance-of link. In delegation-based prototype-based languages such as Self, where an implicit instance-class link is modelled by an explicit prototype-traits link, the programmer is able to intervene. As an extreme case, dSelf [25], a distributed extension of Self, allows the prototype and its traits to be physically distributed such that prototypes can share traits across the network and avoid copying altogether, if this is desirable.

Strengthened by the observation that existing distributed programming languages such as Emerald [14], Obliq [4], dSelf [25] and E [21] are all classless, we state that object-based object models are a solid basis to develop robust software for Ambient Intelligence.

2.2 Fundamental Characteristic Operations of OOP

This section establishes the basic operations of a full-fledged object model, which guide our analysis of the encapsulation problems in existing object-oriented languages. Fundamental to any object-oriented language is the notion of message sending: the ability of objects to send each other polymorphic messages that are implemented by a corresponding method implementation, encapsulated in a receiver object. Additional important object-oriented concepts such as incremental specification of abstractions and object creation differ depending on whether the language is class-based or prototype-based. In order to unify these concepts from both subparadigms, we reuse terminology from an influential taxonomy paper, *The Treaty of Orlando* [17], where Lieberman, Stein and Ungar characterize these concepts based on two fundamental mechanisms: *empathy* and *templates*. In addition, we consider the notion of computational reflection [18], since it is encountered in most mature object-oriented languages.

Templates refer to the entities from which new objects can be spawned by a so-called "cookie-cutting" mechanism. In a class-based language, classes act as templates, and object creation is done by instantiating these classes. In a prototype-based language one can employ any object as a template, by simply cloning the existing object.

Empathy refers to the ability to share behavior between templates, as to specify abstractions in an incremental fashion. Class-based languages achieve this structural reuse through class-based inheritance. In a prototype-based language this reuse is typically achieved through delegation between an object and its parents [16].

Reflection refers to the ability for a program to consult and modify its own high-level structure. In an object-oriented language, this typically implies that an object can acquire access to a reified version of both its data and method slots. Most mature programming languages incorporate a form of reflection for *e.g.* debugging, implementing language extensions, providing object serialization, etc.

These operations are fundamental to object-oriented programming, which is why languages usually provide the programmer with explicit language operators that enable the application of these operations on objects or classes. Whereas none of these *operations* breaches the encapsulation of objects *per se*, their corresponding language *operators* usually do. A notable exception is the message sending operator, which allows the receiver object to decide how it responds to the message. The encapsulation problems engendered by the other operators are illustrated in the following section.

3. LANGUAGE OPERATIONS COMPROMISING ENCAPSULATION

The main motivation for our work on encapsulation for Ambient-Oriented programming is that language operators in contemporary programming languages can breach object encapsulation. In this section, we discuss how ill-designed object extension, cloning and reflection operators threaten the encapsulation of objects.

3.1 Conflicting Operation: Extension

Section 2.2 pinpointed empathy, the ability to incrementally specify abstractions, as an essential characteristic of object-oriented languages. Unfortunately, empathy can only be achieved at the cost of breaching the encapsulation of the object or class that is being extended [22].

Delegation-based prototype-based languages often emphasize flexibility, which hampers the introduction of proper object encapsulation. In contrast with class-based inheritance where private instance variables of the superclass remain inaccessible to the subclass, child prototypes can access their parent's private state. This allows for some expressive programming patterns [16]. The problem is that in languages employing delegation, such as Self, any object can freely designate another object as its parent, thereby breaching the parent's encapsulation.

Some object-based languages –not featuring prototype-based delegation– have attempted to reconcile an object extension mechanism with encapsulation:

- E [21] achieves encapsulation by not providing built-in support for inheritance. Instead E's `extends` keyword specifies that an object simply *forwards* messages it does not understand to a parent object, whose state remains private to itself. Hence, E sacrifices private state sharing via common parents in favor of encapsulation. We will discuss E in more details in Section 6.3.

- Obliq [4] introduces a particular extension mechanism, by embedding the clones of two or more objects in a new object. The resulting object therefore exhibits an extended behavior resulting from the combination of the original objects. This extension mechanism strongly relies on cloning, discussed in the following section.

Incremental specification of objects is an essential ingredient of an object-oriented language. However, existing languages often implement extension using operators external to the object, such that the encapsulation of the object under extension is breached. In order to reconcile extension with flexible object encapsulation, objects should explicitly *cooperate* to be extended.

3.2 Conflicting Operation: Cloning

Most object-oriented languages, either class-based or prototype-based, make it possible to *clone* objects. In a prototype-based language, cloning is essential as it is the fundamental operation to both create new objects as well as new abstractions. Cloning objects without their explicit consent can be a severe security breach. For instance, Obliq [4] provides a `clone` operator that allows an object to clone any object it refers to. In pseudocode, an object `obj` can be cloned as follows:

```
clonedObj = clone(obj);
```

Cardelli recognizes that such an unrestricted cloning in Obliq is potentially hazardous and thus also provides a `protect` operator to fully encapsulate an object: a protected object is shielded from the external application of operators such as cloning, they can only be applied by the object itself. We will return to Obliq in Section 6.2.

A better implementation of the clone operation consists of introducing `clone` methods instead of an external `clone` operator. The idea is that, upon receipt of a `clone` message, an object by default returns a clone of itself. An object can refuse to be cloned or intervene in the cloning process by *e.g.* overriding the `clone` method. Cloning an object is then achieved by executing:

```
clonedObj = obj.clone();
```

The language Self [27] employs this scheme of cloning objects by sending them `clone` messages. Objects which delegate to the `cloneable` trait automatically inherit the default cloning behavior. In the case of Java, for an object to be publicly cloneable, its class must implement the `Cloneable` interface and override the `clone` method (which is protected) with a public method.

Although these solutions make it somehow possible to protect objects from being unexpectedly cloned, there remains a fundamental problem that stems from the use of a cloning operator: the clone initialization has to be done by the object creating the clone, which requires access to the clone's private state. This often requires mutator methods to make the state accessible, which defeats an object's encapsulation. For example:

```
clonedCreditCard = aCreditCard.clone();
clonedCreditCard.setCardNumber(...);
clonedCreditCard.setCardHolder(...);
clonedCreditCard.setExpiryDate(...);
```

In other words, cloning operators must usually be employed in conjunction with external updates of the state of the clone object: this can be a severe breach of object encapsulation, as it may require otherwise unnecessary mutator methods to be publicly provided.

3.3 Conflicting Operation: Reflection

Most mature object-oriented languages also provide reflective provisions, which can be used to access the state of an object in metaprograms. This is typically used for marshaling arguments of a remote invocation, debugging a program, or even for building the IDE of the language. Typically, a reflective API is the major hindrance to preserving encapsulation, because it allows one to breach the integrity of an object.

For example, in Java, any object can obtain a reification of a class (of type `Class`) using the `.class` pseudo-variable. Since `.class` is not a real message send, any class that is publicly available can be accessed through the reflection API, thereby revealing its fields and methods. The use of a security manager based on access control lists (ACLs) solves the issue of controlling reflection, but in a coarse-grained and rigid manner; as argued in [19], security based on *object capabilities* (in essence, unforgeable object references) offers both fine granularity and flexibility.

Mirror-based reflection [3] is essentially compatible with capabilities. The idea of mirrors is that metalevel facilities are not accessed from an object or class directly, as is the case in the Java language for instance, but rather that such facilities are exposed by special objects called *mirrors*. Hence, in this model, the ability to reflect upon an object coincides with the notion of having a reference to (a capability for) a mirror object. This means that reflective access to an object can be regulated depending on the kind of mirrors that are handed out to other objects.

Bracha and Ungar state a number of design principles to which mirror-based reflective systems should adhere. One such principle is *stratification*: the principled separation of base-level and meta-level concerns. One advantage of such stratification is that reflective code can easily be withdrawn from the application when it is not needed. Doing so can considerably reduce the footprint of an application, and so this principle is highly relevant in the context of embedded and ubiquitous computing, where small platforms are the rule.

However, the mirror design as presented in [3] is not compatible as such with object encapsulation. The incompatibility stems from the fact that an object is not involved at all in the process of giving a mirror to itself: rather, a mirror factory is queried:

```
mirror = MirrorFactory.getMirror(obj)
```

This is very similar to the external cloning operator sketched in the previous section. Ideally, an object should have precise control over what mirror is handed out to whom. In particular, it is important that an object is able to hand over a read-only mirror to untrusted clients, and a read-write mirror to trusted ones. Such choices can only be made by actively involving the object in the process of mirror acquisition.

3.4 Conclusion

We have shown how language operators that allow for object extension, cloning and reflection can breach object encapsulation. These encapsulation problems are most severe for delegation-based prototype-based languages, as delegation gives access to parent state and cloning is fundamental to the language paradigm.

The recurring cause of the encapsulation breaches is that the object which is subject to an operation has no means to intervene in the execution of that operation. Objects cannot restrict their extensions, their clones or their mirrors. This observation has lead us to rethink the design of such language operations. The foundations of the resulting object model are the topic of the following section.

4. THE EXTREME ENCAPSULATION PRINCIPLE

The existence of *language operators* that allow programmers to extend, clone or reflect upon any object, without requiring the explicit consent of the object, is the source of encapsulation breaches. In the context of ubiquitous computing, it is crucial that both mobile objects and hosts can protect themselves and avert encapsulation breaches. To ensure object confidentiality and integrity, we define a general language design principle, called *extreme encapsulation* [5].

Extreme Encapsulation – An object can designate some of its internal state and operations to be private and *enforce* this property. To be able to uphold this principle, *language operators* which manipulate an object or class *without its explicit intervention* are to be prohibited.

To proponents of object-oriented programming, following this principle may seem like common sense, as encapsulation is one of the hallmarks of object-oriented programming. However, when looking at contemporary object-oriented languages, we cannot help but notice that most of them break this principle as illustrated in section 3. The ones that do not violate the principle, such as pure actor languages, only do so by disabling language operators entirely, but this often leads to software engineering restrictions.

Given the extreme encapsulation principle, the prime question that arises is how it can be reconciled with the demands of a rich object model characteristic of *e.g.* a prototype-based language, requiring object extension, cloning, and reflection. In each of these cases, the extreme encapsulation principle has its specialized formulation:

Extension Protection – An object can decide *for itself* whether or not it may be extended, and under which restrictions. Failing to do so would allow malicious extensions to posses state and behavior they are not entitled to have.

Cloning Protection – An object can decide *for itself* whether or not it may be cloned, and under which restrictions. Failing to do so would allow malicious objects to make uncontrolled amounts of clones, which would amplify their capabilities.

Reflection Protection – An object can decide *for itself* how much of itself it exposes at the metalevel. Failing to do so would allow malicious meta-programs to circumvent an object's interface completely, exposing an object's implementation at the meta-level.

Therefore, the main point of the extreme encapsulation principle is that even though operations such as extension, cloning and reflection are to be provided, they should be so in a way that *actively involves* the concerned object, rather than being offered as omnipotent language operators. In other words, the extreme encapsulation principle states that:

> *Any operation in a programming language –including extension, cloning, and reflection– should happen through message passing, and message passing alone.*

These design principles actually raise the issue of how to implement a language where message passing alone can be used for traditional communication between objects, as well as for extending, cloning and reflecting upon objects. Our solution to this issue is to consider different interpretation semantics for methods, depending on their associated *method attribute*, as explained and illustrated in the remainder of this paper.

5. METHOD ATTRIBUTES FOR EXTREMELY-ENCAPSULATED PROTOTYPES

Our proposal to reconcile extreme encapsulation with a rich object model hinges on the introduction of *method attributes* to specialize the way methods are evaluated. Each specialization plays the role of a language operator, which is guaranteed only to operate on the provider of the method and not on some external object. Before detailing how method attributes are incorporated in an experimental prototype-based programming language, called ChitChat, we give a language-neutral explanation of method attributes through the use of a semi-formal model based on simple evaluation rules.

5.1 Method Attributes

We employ a semi-formal object-oriented language model that supports three operations: object extension (*i.e.* delegation), object cloning and mirror-based reflection [3]. In order to support these operations while maintaining extremely encapsulated objects, three method attributes are introduced: `create`, `clone` and `mirror`. Methods can be annotated with one or more of these attributes, resulting in *create methods*, *clone methods*, *mirror methods* or a combination thereof.

Attributed methods have a specialized evaluation semantics when they are invoked: they may perform some operations on the receiver both before and after the evaluation of the method body, but most importantly, *they can modify the context in which the method body is evaluated*. In other words, an attributed method may be evaluated in another object than the one that receives the message.

Without loss of generality, we do not consider method arguments as they are not fundamental to the understanding of method attributes. In the same vein, lexical environments are omitted from our semi-formal model as the interaction between lexical scoping and object-oriented concepts is well understood [1]. When considering the abstract grammar for a language supporting method attributes, two language constructs are particularly important, namely the invocation and declaration of methods:

```
send ::= message(receiver, selector)
method ::= create(method)
        |  clone(method)
        |  mirror(method)
        |  basic_method(body)
```

5.1.1 Evaluation rules

Typically, an evaluation rule takes an abstract grammar element and an evaluation environment. The environment holds relevant information such as the current scope, the current `self`, etc. However, as explained above, for the purposes of explaining method attributes, only the `self` binding will be considered. Objects in our model are composed of slots, which can be marked either private or public.

The evaluation of a message send is based on the intuitive *apply* $\circ$ *lookup* (read: apply *after* lookup) scheme of MOOSTRAP [20]. The essence of the evaluation rule is that an object is first queried for a suitable method corresponding to the received message, and then the method is applied to the receiver object. We distinguish self-sends from other message sends, because self-sends may query an object for both private and public slots, while external sends may only query an object for public slots. Message sends are evaluated as follows:

$$
\begin{array}{l}
\text{eval}(message(rcvexp, selector), self) \equiv \\
\quad \textbf{let}\ recv = \text{eval}(rcvexp, self)\ \textbf{in} \\
\quad\ \textbf{let}\ meth = \begin{cases} \text{lookup}_{all}(recv, selector) & \text{if } recv = self \\ \text{lookup}_{pub}(recv, selector) & \text{otherwise} \end{cases} \quad \textbf{in} \\
\qquad\quad \text{apply}(meth, recv)
\end{array}
$$

Due to the introduction of method attributes, apply becomes a polymorphic operation: *there is no longer one single semantics for method application*. Each attribute modifies the method application process. Therefore, apply dispatches on each attribute of the method, giving rise to the following evaluation rules:

$$
\begin{array}{l}
\text{apply}(create(method), self) \equiv \text{apply}(method, \textbf{extend}(self)) \\
\text{apply}(clone(method), self) \equiv \text{apply}(method, \textbf{clone}(self)) \\
\text{apply}(mirror(method), self) \equiv \text{apply}(method, \textbf{reflect}(self)) \\
\text{apply}(basic\_method(body), self) \equiv \text{eval}(body, self)
\end{array}
$$

Method attributes actually *change* the receiver of the method invocation:

- a create method applies on an *extension* of the original receiver, obtained by application of the **extend** operator;
- a clone method applies on a *clone* of the original receiver, obtained by application of the **clone** operator;
- a mirror method applies on a *mirror* object reflecting upon the original receiver, obtained by application of the **reflect** operator.

The essence of method attributes is that they enable operators such as **extend** and **clone** –which are usually explicit in an object-oriented language – to be *hidden* in the semantics of method application, such that it can be *guaranteed* that they are applied exclusively to `self`. Hence, an object can give up on its encapsulation only if it explicitly decides to do so by implementing a method with the proper method attribute. If the operators were explicit in the language, any object could apply them to any other object, which would lead to uncontrolled breaches of encapsulation as discussed in Section 3.

5.1.2 Semantics of operators

Each of the evaluation rules for the method attributes relies on a special operator (**extend**, **clone** and **reflect**) that acts upon `self` and returns an alternative object. In order to describe the behavior of these operators, the structure of objects upon which operators act is defined as follows: an object is represented as a triple $\langle vars, meths, parent\rangle$ consisting of the its state, behavior and delegation link.

Variables and methods are represented as simple association lists of `name=value` pairs. All objects ultimately derive from a predefined `root` object. The three operators used above are defined as follows:

(1) $\mathbf{extend}(self) \equiv \langle \texttt{[]}, \texttt{[]}, self \rangle$
(2) $\mathbf{clone}(\langle vars, meths, par \rangle) \equiv \langle \mathrm{copy}(vars), meths, par \rangle$
(3) $\mathbf{reflect}(self) \equiv \langle \texttt{[meta=}\mathrm{reify}(self)\texttt{]}, \texttt{[]}, root \rangle$

Extending yields an empty object whose parent is the original receiver (1). Cloning yields an object in which the state of the original receiver is copied, and which shares the method implementations and parent of the original receiver (2). Finally, the **reflect** operator yields a mirror, which is an empty object extending `root`, with an extra variable called `meta` that refers to a *reification* of the original receiver (3). Such a reification offers introspective and intercessive facilities to inspect and modify the object at the metalevel. We will give examples of such facilities in Section 5.5.

In summary, the crux of method attributes is that they ensure that possibly harmful operations on an object can only be applied to the provider of an attributed method. Attributes modify the receiver in which the attributed method body will be applied. They only allow for controlled encapsulation breaches: an object can only be extended, cloned or reflected upon if it implements a method with the corresponding method attribute.

5.2 ChitChat: Extremely Encapsulated Prototypes

The model explained so far has been applied to an experimental, distributed prototype-based language ChitChat, whose virtual machine is implemented in Java. ChitChat is a proof by construction that extreme encapsulation can be reconciled with full-fledged object-orientation thanks to the model of method attributes explained in the previous section[1].

Because the syntax of ChitChat is very close to that of Javascript, we exploit their syntactic resemblance by showing how a simple `BankAccount` object is created in both of these languages. The object consists of one variable (`balance`) and two methods (`deposit` and `withdraw`). The Javascript code is shown first, while the ChitChat version is shown immediately below it. The JavaScript code features a so-called *constructor function* that describes the structure of a `BankAccount` object. A concrete `account` object is created by simply calling this function. In ChitChat, the equivalent object structure can be created by defining a *create method* (*i.e.* a method annotated with the `create` attribute). In the example, this method is associated with the top-level `root` object because it is declared in the top level. An `account` object is created by invoking the create method on the `root` object as illustrated on the last line. Note that `deposit` and `withdraw` are normal, unattributed methods.

```
function BankAccount(amount) {
  this.balance = amount;

  this.deposit = function(sum) {
    this.balance = this.balance + sum;
  };
  this.withdraw = function(sum) {
    if (this.balance < sum) {
      alert("insufficient funds");
    } else {
      this.balance = this.balance - sum;
    }
  };
};
account = new BankAccount(20);
```

```
create.BankAccount(amount) :: {
  balance : amount;

  deposit(sum) :: {
    balance := balance + sum;
  };
  withdraw(sum) :: {
    if (balance < sum,
        error("insufficient funds"),
        balance := balance - sum)
  };
};

account : root.BankAccount(20);
```

Because JavaScript aims for flexibility, it allows for all slots of its objects to be read and modified by external objects. Moreover, new fields can be added to existing JavaScript objects. In contrast, ChitChat allows its objects to enforce the privacy of variables and methods. In ChitChat, private variables and methods are defined using `:`, whereas public variables and methods are defined using `::`. Private variables and methods cannot be accessed by external objects.

The object model of ChitChat, as presented up to now, adheres to the extreme encapsulation principle, as the encapsulation of private variables and methods can be enforced. However, we have yet to explain how the addition of object extension, cloning and reflection can be achieved without violating this principle. This is the topic of the following sections.

5.3 Create methods

In the previous section we have illustrated how create methods are used to create new objects. Conceptually, new objects are but extensions of the `root`. Create methods can also be used to create extensions from objects other than the `root`. To uphold extension protection (Sect. 4), an object should itself provide a create method, if it is to be extended. For instance, in order to extend the `BankAccount` abstraction, a nested create method must be created to house the extension:

```
create.BankAccount(amount) :: {
  ... as before ...
  create.LimitedWithdrawAccount(limit) :: {
    withdraw(sum) :: {
      if(sum > limit,
        error("withdrawal limit exceeded"),
        super.withdraw(sum))
    }
  }
}
a : BankAccount(20).
  LimitedWithdrawAccount(5);
a.withdraw(10);
// --> error: withdrawal limit exceeded
```

[1] A slight difference between the semantics of method attributes outlined in Section 5.1 and ChitChat is that, for convenience, attributed ChitChat methods always implicitly return the receiver object in case it was changed: for instance, a create method always returns the newly-created extension. Without such a convenience, each attributed method would need to explicitly return `self`.

The `LimitedWithdrawAccount` extension overrides the `withdraw` method to ensure that a client can only withdraw a limited sum per transaction. Inside the body of this method, the extension can refer to its parent object using `super`.

5.3.1 Unanticipated extension

In our model, all extension possibilities have to be anticipated, since they are expressed as nested create methods. This may seem to be a limitation, because in some cases, one would like to allow for unanticipated extension. Although we have argued that such extensions are a potential breach of encapsulation, it is possible to express them in ChitChat. In this case, an object explicitly sacrifices its encapsulation for the sake of being extensible in any manner. In essence, an object needs to implement a create method taking a quoted block of code as an argument and then evaluate this code: the quoted code fulfills the role of the create method body, but can now be provided externally.

```
create.Person(aName) :: {
  name : aName;
  toString() :: { name };
  create.extend(extensionCode()) :: {
    extensionCode() }
};
Doctor(aName) :: Person(aName).extend({
     toString() :: {
       "Dr. " + super.toString() }
})
```

In this example, a `Person` abstraction offers an `extend` create method that evaluates any piece of code in the environment of the extension[2]. This makes it possible to add more specialized `Person` abstractions later on. In the example, `Doctor` objects extend `Person` objects, overriding the `toString` method.

Such unanticipated extensions are still in accordance with the extreme encapsulation principle, since the object itself explicitly grants extensibility. Extreme encapsulation does not prohibit encapsulation breaches, rather it prohibits *unwanted* ones.

5.4 Clone methods

Clone methods allow an object to serve as a template for creating new objects by copy, rather than serving as an extensible parent by providing a create method. Whereas cloning itself may not be a breach of object encapsulation, the use of a clone operator (*e.g.* `clone(account)`) implies that the correct re-initialization of the clone should be performed by the object that clones the account (*e.g.* the client). In order to perform such initialization, the encapsulation of the clone would need to be broken (as explained in Section 3.2). Clone methods provide a valuable alternative, since the initialization code of the clone is specified in the body of the clone method.

For instance, to allow secure cloning of bank accounts (where private, instance-specific information should not be copied), the following clone method could be defined inside the `BankAccount` create method:

```
create.BankAccount(amount) :: {
  /* as before */
  clone.new(newAmount) :: {
    balance := newAmount; }
}
```

[2]When the formal parameter of a ChitChat method is affixed with parentheses, the argument passed is quoted. The details regarding this quoting mechanism are explained in [7].

The assignment statement in the body of the clone method affects the `balance` variable of the clone instead of the variable of the original receiver. Clone methods can also be used to ensure that sensitive objects –*e.g.* digital money– cannot be cloned with wrong intents, for instance by requesting some certificates to be passed as argument:

```
create.EMoney(amount) :: {
  /* variables and methods */
  // doCopy() is private due to :
  clone.doCopy() : { ... };
  copy(certificate) :: {
    if(isValid(certificate),
       doCopy(),
       error("invalid certificate")) } }
```

The clone method `doCopy` is *private* (defined with `:` rather than `::`), and hence cannot be invoked by external objects. The `copy` method is public and expects a certificate object as parameter: if the certificate is valid, then `doCopy` is called, resulting in a clone being returned to the caller, otherwise an error occurs.

Finally, it is easy to ensure in ChitChat that an object is never cloned: it suffices *not* to implement any clone method at all.

5.5 Mirror methods

ChitChat offers facilities for reflective programming via *mirror methods*. Recall that a mirror method is evaluated by first creating a mirror of the receiver and evaluating the method body *in the context of the mirror*. The return value of a mirror method is the mirror itself. A mirror object is a special object that relates to its creator in the sense that it has an implicitly initialized `meta` variable referring to a reification of its creator. An important rule is that *only mirrors have access to the reification of their creator*. Furthermore, this access is restricted by the creator of the mirror. Apart from this, mirrors are normal objects: they can be sent messages and, if they procure the possibilities to do so, they can be cloned, extended, etc. To respect the principle of reflection protection, mirrors can only be obtained via mirror methods explicitly defined by the object to be mirrored.

The interface of the `meta` variable determines the actual *metaobject protocol* [15] that is available. This protocol depends on how open the ChitChat interpreter is. Basic facilities include access to the structure and behavior of an object. An open virtual machine may expose more facilities: for instance, if the VM relies on reference counting for garbage collection, it may offer the possibilities to notify a mirror each time the reference count of the base object is increased or decreased. Here, we only mention a basic metalevel API offering access to the behavior and structure of objects.

5.5.1 Structural API

The `meta` variable gives access to the different structural elements of an object: variables, methods and the parent object. However, in order to respect the stratification principle explained in Section 3.3, variables and methods are not returned as their direct metalevel implementation objects, but rather as mirrors to these objects. There are different variants of mirrors, offering different degrees of accessibility. For space reasons, we do not consider mirror access rights in this paper. More details on mirror methods can be found in [24].

A mirror giving access to the reification of a method is obtained by calling `meta.method(m)` where `m` is the name of the method to be mirrored. Alternatively, one can use `meta.methods()` to obtain an array of such mirror objects for all the methods of the receiver.

```
create.BankAccount(amount) :: {
  /* as before */
  mirror.invoker() :: { meta.methods() }
}
```

When invoked, the `invoker` mirror method returns a mirror of the receiver bank account. The mirror has one method, `methods`. Since the mirror `meta` variable refers to a reification of the receiver, `meta.methods()` refers to the method table of the receiver. Hence, if `a` is a bank account, `a.invoker().methods()` returns the table of all the methods of `a`. One can then use this table to introspect and invoke methods on `a` reflectively. An equivalent API is available for fields, via `meta.field(f)` and `meta.fields()`.

This API satisfies the reflection protection principle: reflective access to an object's fields and methods requires the object's explicit cooperation, as they can only be accessed via message passing.

5.5.2 Behavioral API

Mirror methods can also expose the behavior of an object. Monitoring the behavior of an object is made possible by registering *listeners*. A mirror can register itself as a listener of meta-level events (method calls, field accesses) of its base object.

```
create.BankAccount(amount) :: {
  /* as before */
  mirror.balanceListener(actionMethod) :: {
    onReturn() :: {
      balance : meta.field("balance");
      actionMethod(balance.read())
    };
    meta.after(self,
      ["deposit", "withdraw"],
      self.onReturn)
  }
}
```

The `balanceListener` mirror method above makes it possible to specify that an action has to be performed after changes in the balance of the account object. More precisely, the mirror registers itself as a listener to be notified *after* invocations of the `deposit` and `withdraw` methods, with `meta.after(..)`. The `onReturn` method of the mirror object is called for notification: this method reflectively accesses the value of the `balance` field of the original account, and applies the `actionMethod` given as parameter, passing it the current value of the balance. A usage example of this mirror method is given below:

```
create.AccountStore(account) :: {
  write(amount) :: {
    /* write account balance on disk */
  };
  account.balanceListener(self.write)
}
a : BankAccount(20);
a.withdraw(5);
// --> no storage update
s : AccountStore(a);
a.deposit(2);
// --> storage update
```

An `AccountStore` object is used to store the balance of an account to disk. When created, this object invokes the mirror method passing it as parameter the `write` method that should be invoked: this method does the actual job of writing to the disk.

As the example shows, mirror methods allow an object to decide exactly which part of the interface of its reification should be accessible to other objects. In the example, an account object only allows intervention upon state changes. The object also has control over the design of the reflective interface exposed to clients (in the example, `balanceListener`). This makes it possible to cleanly encapsulate implementation details: the `AccountStore` does not need to know that updates to the balance occur when `withdraw` or `deposit` are called. Mirror methods hence completely satisfy the constraint imposed by the *reflection protection* principle, as they ensure that an object can precisely describe to what extent it wishes to protect itself against metalevel operations.

6. RELATED WORK

In this section we review related proposals to ameliorate object encapsulation, in particular Agora [23], since it is the ancestor of ChitChat, and Obliq [4], as it introduces the notion of *self-inflicted operations*, which achieves a level of object encapsulation similar to extreme encapsulation. Finally, we discuss E [21], a modern distributed prototype-based language that emphasizes language-level security.

6.1 Agora: Mixin Methods

The foundations of the object model of ChitChat are the outcome of previous research on prototype-based object models [10, 6]. In particular, the predecessor of ChitChat, Agora [23], first introduced the use of so-called *mixin* methods to extend objects. Agora was the first language to adhere to the extreme encapsulation principle: since Agora objects implement their own mixin methods, they can determine for themselves whether and how much they can be extended. The mixin methods of Agora correspond to the create methods of ChitChat. The contribution of ChitChat is to extend this notion to other language operations like cloning and mirror-based reflection. Furthermore, ChitChat specifically targets dynamic and open networks by proposing a model for concurrency, distribution and mobility. This model is not presented in this paper for space reasons, but the interested reader can refer to [5].

6.2 Obliq: Protected Objects

Obliq [4] is a distributed object-based language with support for mobility. Obliq provides four operations applicable to objects: field selection (method invocation), field update (method overriding), cloning and aliasing. The operations are incorporated in the language in the form of operators which can be externally applied to an object. The first operator corresponds to the message sending operation of ChitChat. The field update operator allows for objects to be externally modified, as follows:

```
o.x := value
```

The cloning operator is used to create a new object by concatenating the clones of one or more objects. By cloning more than one object, the operator encompasses extension of existing objects with new behavior, as is shown in the following example:

```
    aColoredShape := clone(aColor, aShape)
```

Finally, aliasing is an operator which allows operations on an object's field to be redirected to another object's field. There is no such equivalent operation in ChitChat. Note that the field update, cloning and aliasing operators are potentially harmful as they can be applied to objects without their cooperation.

In order to deal with the potential encapsulation breaches, Cardelli distinguishes two ways in which an object can be subject to one of the four operators: they can be performed either as external operations on an object, or as internal operations through `self`. When a method operates on its own self, the operation is said to be *self-inflicted* [4]. In order to exploit this important difference in operator usage, objects can be declared `protected`. A protected object does not allow external update, cloning, and aliasing operations, but permits such operations when they are self-inflicted. In other words, they are protected from changes made by external objects, but they can still change themselves.

Obliq's protected objects and ChitChat's method attributes share the same goal: protecting an object from uncontrolled external modifications, while still allowing the object to be modified if it offers the possibilities to do so. ChitChat's notion of `create` and `clone` methods can be expressed in Obliq by using the standard operators via self-sends. There is no equivalent in Obliq for ChitChat's `mirror` methods, however.

6.3 E: Secure Distributed Prototypes

To the best of our knowledge, the only other well-known object-oriented language for distribution that aims at strong encapsulation is the E programming language [21]. E has many similarities with ChitChat, particularly with respect to its distribution model (not presented in this paper, but which can be found in [5]). However, E achieves strong encapsulation by adopting an object model that is reduced to the bare minimum: in this sense it is not considered to be a prototype-based language, as it does not feature true delegation.

The object model employed by E is strongly rooted in the lambda calculus. In essence, E objects are closures around lexical environments. Additionally, E provides the necessary language constructs to support the use of public methods, message passing and private instance variables.

Although the object model of E is sufficiently expressive to support many object-oriented applications, it is less expressive than object models employing classes or prototypes. This becomes apparent when more advanced object-oriented features, such as delegation or inheritance, are required. E does not support prototype-based delegation (with late binding of `self`), it only supports implicit forwarding of messages to parents. To *simulate* late binding of `self`, an E programmer must use the following pattern, whereby code is explicitly parameterized with a `self` variable:

```
# extensions should provide self
def makeCollection(self) {
  def abstractCollection {
    to map(f) {
      # using self parameter
      var it := self.iterator();
      while(it.hasNext()){
        f(it.next())
      }
    }
  }
  return abstractCollection }

def makeList() {
  # child is explicitly passed as self
  def list extends makeCollection(list) {
    to iterator(){
       # return a list iterator
    }
  }
  return list }
```

As can be seen in the code, the `self` variable usually implicitly managed by the language has to be explicitly passed around by the programmer. It has to be noted that the inheritance scheme simulated by this pattern is similar to class-based inheritance because there is no parent sharing: when the list is created, its parent is a newly-created and private collection object. But there is still a fundamental difference between E and real class-based inheritance: in E, calling `foo()` or `self.foo()` do not have the same semantics (simply calling `foo()` will not entail late binding of `self`, hence `foo` will be looked up in the environment of the parent only).

The major contribution of ChitChat is to promote extreme encapsulation in a prototype-based language (featuring prototype-based delegation that enables parent sharing). Furthermore, E does not support cloning: new objects are always obtained by invoking generator functions, which are similar to the create methods of ChitChat. With respect to reflection, E has a limited number of introspective facilities. The native functions `E.send` and `E.call` can be used to reflectively invoke public methods of E objects. Moreover, `E.match` can be used to trap messages that are not understood by an object, and as such supports the same metalevel programming techniques enabled by `doesNotUnderstand` in Smalltalk [11]. By providing only this limited set of introspective facilities, E adheres to our *reflection protection* principle. This is achieved only because reflection is basically ruled out. Conversely, in ChitChat, an object can unveil much more than only its public interface at the metalevel, but only if it wants to.

7. CONCLUSION

When considering mobile computing applications in an Ambient Intelligence context, the devices forming ad hoc wireless networks are discovered dynamically as the user moves about. Dynamically encountered devices cannot always be trusted, which is why untrusted objects can mingle with trusted ones in the mobile object system. If mobile objects are to remain useful in this context, one must be able to shield objects from external harm. We have explored the principled enforcement of object encapsulation as a means to do so.

We have introduced the *extreme encapsulation* principle, which states that an object should remain in total control over all operations applied to it. This is achieved by providing all language operations by means of message sending only. We have devised a prototype-based language, ChitChat, which provides a full-fledged object model while respecting the extreme encapsulation principle. The key insight is a language feature called *method attributes*, which makes it possible for methods to have different interpretation semantics: create methods enable object extension with true delegation, clone methods enable controlled object cloning, and mirror methods enable selective reflective access to the state and behavior of objects.

In parallel with the design and implementation of ChitChat, we have constructed a distribution model specifically tailored towards ubiquitous computing. This distribution model, called the *ambient actor model* [8] is an extension of the well-known actor model of computation [2]. An interesting perspective for future research is the incorporation of method attributes in this distribution model, thereby making it adhere to the extreme encapsulation principle. To this end, we need to identify harmful operators that arise in concurrent and distributed programming languages and to convert them into method attributes, exploring how this change in design benefits the development of applications for ubiquitous computing.

Acknowledgements
We thank the anonymous referees for their valuable clarifications, comments and critiques on earlier versions of the paper.

8. REFERENCES

[1] ABADI, M., AND CARDELLI, L. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] AGHA, G. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] BRACHA, G., AND UNGAR, D. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications* (2004), pp. 331–343.

[4] CARDELLI, L. A Language with Distributed Scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Press, pp. 286–297.

[5] DE MEUTER, W. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, 2004.

[6] DE MEUTER, W., D'HONDT, T., AND DEDECKER, J. Intersecting classes and prototypes. In *5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia* (2003), vol. 2890 of *Lecture Notes in Computer Science*, Springer.

[7] DE MEUTER, W., D'HONDT, T., AND DEDECKER, J. Pico: Scheme for mere mortals. In *ECOOP Workshops* (2004), J. Malenfant and B. M. Østvold, Eds., vol. 3344 of *Lecture Notes in Computer Science*, Springer.

[8] DEDECKER, J., AND VAN BELLE, W. Actors for Mobile Ad-hoc Networks. In *International Conference on Embedded and Ubiquitous Computing EUC2004* (August 2004), L. Yang, M. Guo, G. Gao, and N. Jha, Eds., vol. 3207 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 482–494.

[9] DEDECKER, J., VAN CUTSEM, T., MOSTINCKX, S., DE MEUTER, W., AND D'HONDT, T. AmbientTalk: A Small Reflective Kernel for Programming Mobile Network Applications. Tech. rep., Vrije Universiteit Brussel, 2005.

[10] D'HONDT, T., AND DE MEUTER, W. Of first-class methods and dynamic scope. *RSTI - L'objet no. 9/ 2003. LMO 2003* (2003), 137–149.

[11] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[12] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. GOTOP Information Inc., 1996.

[13] IST ADVISORY GROUP Ambient intelligence: from vision to reality, September 2003.

[14] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.

[15] KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[16] LIEBERMAN, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (1986), ACM Press, pp. 214–223.

[17] LIEBERMAN, H., STEIN, L., AND UNGAR, D. Treaty of orlando. In *Addendum to the proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)* (1987), ACM Press, pp. 43–44.

[18] MAES, P. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (New York, NY, USA, 1987), ACM Press, pp. 147–155.

[19] MILLER, M., YEE, K., AND SHAPIRO, J. Capability myths demolished. Tech. rep., Combex, Inc., 2003.

[20] MULET, P., AND COINTE, P. Definition of a reflective kernel for a prototype-based language. In *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan*, S. Nishio and A. Yonezawa, Eds. Springer-Verlag, Berlin, 1993, pp. 128–144.

[21] OPEN E PROJECT. E: Open Source Distributed Capabilities, 2005. `http://www.erights.org`.

[22] SNYDER, A. Encapsulation and Inheritance in Object-oriented Programming Languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (1986), ACM Press, pp. 38–45.

[23] STEYAERT, P., AND DE MEUTER, W. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95* (August 1995), vol. 952 of *Lecture Notes in Computer Science*, Springer, pp. 127–144.

[24] TANTER, E. Mirror methods — reconciling reflection and extreme encapsulation. In *ECOOP Workshop on Object Technology for Ambient Intelligence* (July 2005).

[25] TOLKSDORF, R., AND KNUBBEN, K. Programming Distributed Systems with the Delegation-based Object-oriented Language dSelf. In *Proceedings of the 2002 ACM Symposium on Applied Computing* (2002), ACM Press, pp. 927–931.

[26] UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HÖLZLE, U. Organizing programs without classes. *Lisp Symb. Comput. 4*, 3 (1991), 223–242.

[27] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (1987), ACM Press, pp. 227–242.

[28] VITEK, J., SERRANO, M., AND THANOS, D. Security and Communication in Mobile Object Systems. In *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag: Heidelberg, Germany, 1997, pp. 177–200.

[29] WEISER, M. The computer for the twenty-first century. *Scientific American* (september 1991), 94–100.